---

title: Block Building description: How you can use SUAVE to build blocks on other chains

---

Block Building

One of the unique features that SUAVE enables is block construction for other chains. Whether you're working on constructing transactions, creating bundles, or building blocks, this feature gives you the capability to interact with the latest blockchain state.

Practical Example

Here is an example of a smart contract that demonstrates the practical application of a block building session:

```solidity function sessionExample(bytes memory subTxn, bytes memory subTxn2) public payable returns (bytes memory) { string memory id = Suave.newBuilder();

~~~
Suave.SimulateTransactionResult memory sim1 = Suave.simulateTransaction(id, subTxn);
require(sim1.success == true);
require(sim1.logs.length == 1);

// Simulate the same transaction again should fail due to nonce repetition
Suave.SimulateTransactionResult memory sim2 = Suave.simulateTransaction(id, subTxn);
require(sim2.success == false);

// Simulate the transaction with the correct nonce
Suave.SimulateTransactionResult memory sim3 = Suave.simulateTransaction(id, subTxn2);
require(sim3.success == true);
require(sim3.logs.length == 2);

return abi.encodeWithSelector(this.emptyCallback.selector);
~~~

} ```

In this example, a new builder session is created, and multiple transactions are simulated with varying conditions in order to illustrate how you might approach building blocks with SUAVE.

## Interface

SUAVE exposes several precompiles to help you with transaction simulation and block construction.

If your SUAPP is intended to produce blocks, be they partial or full, you'll first need to start a new builder session. The precompile this calls in the suave-std library looks like this:

`solidity function newBuilder() internal view returns (string memory)`

This function starts a new builder instance within a Kettle. The basic idea is that session ids (the`string` returned by the `newBuilder()` precompile) provide programmatic control when building blocks, one outcome of which is simulating transactions more efficiently. Opening a session enables you to build blocks iteratively, rather than having to re-run all your simulations each time you receive a new transaction or bundle.

:::info

If you'd prefer to just read how the builder is implemented in`suave-geth`, you can **do so here**.

:::

Once you receive transactions or bundles, it is often the case that you need to simulate the effect they will have on the state of the target chain for which your SUAPP is building blocks. You'll often want to construct blocks in stages, as your SUAPP receive various different bundles and/or transactions from different users. This can be achieved with either of the below precompiles:

`solidity function simulateBundle(bytes memory bundleData) internal view returns (uint64) function simulateTransaction(string memory sessionid, bytes memory txn) internal view returns (SimulateTransactionResult memory)`

If you're happy with the results of your simulation and wish to build a block based on the bundles/transactions you've received, you can use one more precompile:

`solidity function buildEthBlock(BuildBlockArgs memory blockArgs, DataId dataId, string memory namespace) internal view returns (bytes memory, bytes memory)`

As the name suggests, we only support building blocks on Ethereum L1 for now. This will change as SUAVE matures.
```

All of these functions utilize the SUAVE Execution Namespace. To understand more about this, please consult the Execution Namespace specification.

Bundles

Bundles are one or more transactions that are grouped together and executed in the order they are provided and are the core unit of block building. If you're unfamiliar with bundles, and want to learn more, you can read this document.

To see how to handle bundles in your SUAPP, check out theSUAVE-STD bundle protocol contract.